Workgroup: Network Working Group
Internet-Draft:
draft-ietf-privacypass-protocol-07
Published: 8 December 2022
Intended Status: Informational
Expires: 11 June 2023
Authors:
S. Celi, Brave Software
A. Davidson, Brave Software
A. Faz-Hernandez, Cloudflare
S. Valdez, Google LLC
C. A. Wood, Cloudflare

Privacy Pass Issuance Protocol

Abstract

This document specifies two variants of the the two-message issuance protocol for Privacy Pass tokens: one that produces tokens that are privately verifiable using the issuance private key, and another that produces tokens that are publicly verifiable using the issuance public key.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction
2. Terminology
3. Configuration
4. Token Challenge Requirements
5. Issuance Protocol for Privately Verifiable Tokens
   5.1. Client-to-Issuer Request
   5.2. Issuer-to-Client Response
   5.3. Finalization
   5.4. Token Verification
   5.5. Issuer Configuration
6. Issuance Protocol for Publicly Verifiable Tokens
   6.1. Client-to-Issuer Request
   6.2. Issuer-to-Client Response
   6.3. Finalization
   6.4. Token Verification
   6.5. Issuer Configuration
7. Security considerations
8. IANA considerations
   8.1. Well-Known 'token-issuer-directory' URI
   8.2. Token Type
   8.3. Media Types
      8.3.1. "application/private-token-request" media type
      8.3.2. "application/private-token-response" media type
9. References
   9.1. Normative References
   9.2. Informative References
Appendix A. Acknowledgements
Appendix B. Test Vectors
   B.1. Issuance Protocol 1 - VOPRF(P-384, SHA-384)
   B.2. Issuance Protocol 2 - Blind RSA, 2048
Authors' Addresses

1. Introduction

The Privacy Pass protocol provides a privacy-preserving authorization mechanism. In essence, the protocol allows clients to provide cryptographic tokens that prove nothing other than that they have been created by a given server in the past [ARCHITECTURE].

This document describes the issuance protocol for Privacy Pass built on [HTTP]. It specifies two variants: one that is privately verifiable using the issuance private key based on the oblivious pseudorandom function from [OPRF], and one that is publicly verifiable using the issuance public key based on the blind RSA signature scheme [BLINDRSA].

This document does not cover the Privacy Pass architecture, including choices that are necessary for ensuring that client privacy leaks. This information is covered in [ARCHITECTURE].

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used throughout this document.

* Client: An entity that provides authorization tokens to services across the Internet, in return for authorization.

* Issuer: A service that produces Privacy Pass tokens to clients.

* Private Key: The secret key used by the Issuer for issuing tokens.

* Public Key: The public key used by the Issuer for issuing and verifying tokens.

Unless otherwise specified, this document encodes protocol messages in TLS notation from [TLS13], Section 3. Moreover, all constants are in network byte order.

3. Configuration

Issuers MUST provide two parameters for configuration:

1. Issuer Request URI: A token request URL for generating access tokens. For example, an Issuer URL might be https://issuer.example.net/example-token-request. This parameter uses resource media type "text/plain".

2. Issuer Public Key values: An Issuer Public Key for an issuance protocol.

The Issuer parameters can be obtained from an Issuer via a directory object, which is a JSON object [RFC8259], Section 4 whose values are other JSON values [RFC8259], Section 3 for the parameters.

Field Name	Value
issuer-request-uri	Issuer Request URI resource percent-encoded URL string, represented as a JSON string [RFC8259], Section 7
token-keys	List of Issuer Public Key values, each represented as JSON objects [RFC8259], Section 4

Table 1

Each "token-keys" JSON object contains the following fields and corresponding raw values.

Field Name	Value
token-type	Integer value of the Token Type, as defined in Section 8.2, represented as a JSON number [RFC8259], Section 6

Field Name	Value
token-key	The base64url encoding of the public key for use with the issuance protocol, including padding, represented as a JSON string [RFC8259], Section 7

Table 2

Issuers MAY advertise multiple token-keys for the same token-type to support key rotation. In this case, Issuers indicate preference for which token key to use based on the order of keys in the list, with preference given to keys earlier in the list.

Altogether, the Issuer's directory could look like:

```
{
   "issuer-request-uri": "https://issuer.example.net/example-token-requ
   "token-keys": [
     {
       "token-type": 2,
       "token-key": "MI...AB",
     },
     {
       "token-type": 2,
       "token-key": "MI...AQ",
     }
   ]
}
```

Issuer directory resources have the media type "application/json" and are located at the well-known location /.well-known/token-issuer-directory; see Section 8.1 for the registration information for this well-known URI.

4. Token Challenge Requirements

Clients receive challenges for tokens, as described in [AUTHSCHEME]. The basic token issuance protocols described in this document can be interactive or non-interactive, and per-origin or cross-origin.

5. Issuance Protocol for Privately Verifiable Tokens

The Privacy Pass issuance protocol is a two message protocol that takes as input a TokenChallenge from the redemption protocol [AUTHSCHEME], Section 2.1 and produces a Token [AUTHSCHEME], Section 2.2, as shown in the figure below.

```
   Origin            Client                      Issuer
                      (pkI)                    (skI, pkI)
                    +------------------------------------\
TokenChallenge ----> TokenRequest ------------->          |
                    |                         (evaluate)  |
     Token     <----+     <---------------- TokenResponse |
                    \------------------------------------/
```

Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 5.5 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

* Issuer name, identifying the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

* Issuer Public Key pkI, with a key identifier token_key_id computed as described in Section 5.5.

* Challenge value challenge, an opaque byte string. For example, this might be provided by the redemption protocol in [AUTHSCHEME].

Given this configuration and these inputs, the two messages exchanged in this protocol are described below. This section uses notation described in [OPRF], Section 4, including SerializeElement and DeserializeElement, SerializeScalar and DeserializeScalar, and DeriveKeyPair.

5.1. Client-to-Issuer Request

The Client first creates a context as follows:

```
client_context = SetupVOPRFClient(0x0004, pkI)
```

Here, 0x0004 is the two-octet identifier corresponding to the OPRF(P-384, SHA-384) ciphersuite in [OPRF]. SetupVOPRFClient is defined in [OPRF], Section 3.2.

The Client then creates an issuance request message for a random value nonce with the input challenge and Issuer key identifier as described below:

```
nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0x0001, nonce, challenge_digest, token_key_id)
blind, blinded_element = client_context.Blind(token_input)
```

The Blind function is defined in [OPRF], Section 3.3.2. If the Blind function fails, the Client aborts the protocol. The Client stores the nonce and challenge_digest values locally for use when finalizing the issuance protocol to produce a token (as described in Section 5.3).

The Client then creates a TokenRequest structured as follows:

```
struct {
   uint16_t token_type = 0x0001; /* Token type VOPRF(P-384, SHA-384) */
   uint8_t truncated_token_key_id;
   uint8_t blinded_msg[Ne];
} TokenRequest;
```

The structure fields are defined as follows:

* "token_type" is a 2-octet integer, which matches the type in the challenge.

* "truncated_token_key_id" is the least significant byte of the token_key_id in network byte order (in other words, the last 8 bits of token_key_id).

* "blinded_msg" is the Ne-octet blinded message defined above, computed as SerializeElement(blinded_element). Ne is as defined in [OPRF], Section 4.

The values token_input and blinded_element are stored locally and used later as described in Section 5.3. The Client then generates an HTTP POST request to send to the Issuer, with the TokenRequest as the content. The media type for this request is "application/private-token-request". An example request is shown below.

```
:method = POST
:scheme = https
:authority = issuer.example.net
:path = /example-token-request
accept = application/private-token-response
cache-control = no-cache, no-store
content-type = application/private-token-request
content-length = <Length of TokenRequest>

<Bytes containing the TokenRequest>
```

Upon receipt of the request, the Issuer validates the following conditions:

* The TokenRequest contains a supported token_type.

* The TokenRequest.truncated_token_key_id corresponds to the truncated key ID of a Public Key owned by the issuer.

* The TokenRequest.blinded_msg is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the client.

5.2. Issuer-to-Client Response

Upon receipt of a TokenRequest, the Issuer tries to deseralize TokenRequest.blinded_msg using DeserializeElement from Section 2.1 of [OPRF], yielding blinded_element. If this fails, the Issuer MUST return an HTTP 400 error to the client. Otherwise, if the Issuer is willing to produce a token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

```
server_context = SetupVOPRFServer(0x0004, skI, pkI)
evaluate_element, proof = server_context.Evaluate(skI, blinded_element)
```

SetupVOPRFServer is in [OPRF], Section 3.2 and Evaluate is defined in [OPRF], Section 3.3.2. The Issuer then creates a TokenResponse structured as follows:

```
struct {
   uint8_t evaluate_msg[Ne];
   uint8_t evaluate_proof[Ns+Ns];
} TokenResponse;
```

The structure fields are defined as follows:

* "evaluate_msg" is the Ne-octet evaluated messaged, computed as SerializeElement(evaluate_element).

* "evaluate_proof" is the (Ns+Ns)-octet serialized proof, which is a pair of Scalar values, computed as concat(SerializeScalar(proof[0]), SerializeScalar(proof[1])), where Ns is as defined in [OPRF], Section 4.

The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".

```
:status = 200
content-type = application/private-token-response
content-length = <Length of TokenResponse>

<Bytes containing the TokenResponse>
```

5.3. Finalization

Upon receipt, the Client handles the response and, if successful, deserializes the content values TokenResponse.evaluate_msg and TokenResponse.evaluate_proof, yielding evaluated_element and proof. If deserialization of either value fails, the Client aborts the protocol. Otherwise, the Client processes the response as follows:

```
authenticator = client_context.Finalize(token_input, blind, evaluated_el
```

The Finalize function is defined in [OPRF], Section 3.3.2. If this succeeds, the Client then constructs a Token as follows:

```
struct {
    uint16_t token_type = 0x0001; /* Token type VOPRF(P-384, SHA-384) */
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

The Token.nonce value is that which was sampled in Section 5.1. If the Finalize function fails, the Client aborts the protocol.

5.4. Token Verification

To verify a token, a verifier creates a VOPRF context using the Issuer Private Key, evaluates the token contents, and compares the result against the token authenticator value, as follows:

```
server_context = SetupVOPRFServer(0x0004, skI, pkI)
token_authenticator_input =
  concat(Token.token_type,
         Token.nonce,
         Token.challenge_digest,
         Token.token_key_id)
token_authenticator = server_context.Evaluate(token_authenticator_input)
valid = (token_authenticator == Token.authenticator)
```

5.5. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. A RECOMMENDED method for generating key pairs is as follows:

```
seed = random(Ns)
(skI, pkI) = DeriveKeyPair(seed, "PrivacyPass")
```

The key identifier for a public key pkI, denoted token_key_id, is computed as follows:

```
token_key_id = SHA256(concat(0x0001, SerializeElement(pkI)))
```

Since Clients truncate token_key_id in each TokenRequest, Issuers should ensure that the truncated form of new key IDs do not collide with other truncated key IDs in rotation.

6. Issuance Protocol for Publicly Verifiable Tokens

This section describes a variant of the issuance protocol in Section 5 for producing publicly verifiable tokens. It differs from the previous variant in that the output tokens are publicly verifiable by anyone with the Issuer public key.

This means any Origin can select a given Issuer to produce tokens, as long as the Origin has the Issuer public key, without explicit coordination or permission from the Issuer. This is because the Issuer does not learn the Origin that requested the token during the issuance protocol.

Beyond this difference, the publicly verifiable issuance protocol variant is nearly identical to the privately verifiable issuance protocol variant. In particular, Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 6.5 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

* Issuer name, identifying the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

* Issuer Public Key pkI, with a key identifier token_key_id computed as described in Section 6.5.

* Challenge value challenge, an opaque byte string. For example, this might be provided by the redemption protocol in [AUTHSCHEME].

Given this configuration and these inputs, the two messages exchanged in this protocol are described below.

6.1. Client-to-Issuer Request

The Client first creates an issuance request message for a random value nonce using the input challenge and Issuer key identifier as follows:

```
nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0x0002, nonce, challenge_digest, token_key_id)
blinded_msg, blind_inv = rsabssa_blind(pkI, token_input)
```

The rsabssa_blind function is defined in [BLINDRSA], Section 5.1.1.. The Client stores the nonce and challenge_digest values locally for use when finalizing the issuance protocol to produce a token (as described in Section 6.3).

The Client then creates a TokenRequest structured as follows:

```
struct {
   uint16_t token_type = 0x0002; /* Token type Blind RSA (2048-bit) */
   uint8_t truncated_token_key_id;
   uint8_t blinded_msg[Nk];
} TokenRequest;
```

The structure fields are defined as follows:

* "token_type" is a 2-octet integer, which matches the type in the challenge.

* "truncated_token_key_id" is the least significant byte of the token_key_id in network byte order (in other words, the last 8 bits of token_key_id).

* "blinded_msg" is the Nk-octet request defined above.

The Client then generates an HTTP POST request to send to the Issuer, with the TokenRequest as the content. The media type for this request is "application/private-token-request". An example request is shown below, where Nk = 512.

```
:method = POST
:scheme = https
:authority = issuer.example.net
:path = /example-token-request
accept = application/private-token-response
cache-control = no-cache, no-store
content-type = application/private-token-request
content-length = <Length of TokenRequest>

<Bytes containing the TokenRequest>
```

Upon receipt of the request, the Issuer validates the following conditions:

* The TokenRequest contains a supported token_type.

* The TokenRequest.truncated_token_key_id corresponds to the truncated key ID of a Public Key owned by the issuer.

* The TokenRequest.blinded_msg is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the Client, which will forward the error to the client.

6.2. Issuer-to-Client Response

If the Issuer is willing to produce a token token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

```
blind_sig = rsabssa_blind_sign(skI, TokenRequest.blinded_msg)
```

This is encoded and transmitted to the client in the following TokenResponse structure:

```
struct {
   uint8_t blind_sig[Nk];
} TokenResponse;
```

The rsabssa_blind_sign function is defined in [BLINDRSA], Section 5.1.2.. The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".

```
:status = 200
content-type = application/private-token-response
content-length = <Length of TokenResponse>

<Bytes containing the TokenResponse>
```

6.3. Finalization

Upon receipt, the Client handles the response and, if successful, processes the content as follows:

```
authenticator = rsabssa_finalize(pkI, nonce, blind_sig, blind_inv)
```

The rsabssa_finalize function is defined in [BLINDRSA], Section 5.1.3.. If this succeeds, the Client then constructs a Token as described in [AUTHSCHEME] as follows:

```
struct {
    uint16_t token_type = 0x0002; /* Token type Blind RSA (2048-bit) */
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

The Token.nonce value is that which was sampled in Section 5.1. If the rsabssa_finalize function fails, the Client aborts the protocol.

6.4. Token Verification

To verify a token, a verifier checks that Token.authenticator is a valid signature over the remainder of the token input using the Issuer Public Key. The function RSASSA-PSS-VERIFY is defined in Section 8.1.2 of [RFC8017], using SHA-384 as the Hash function, MGF1 with SHA-384 as the PSS mask generation function (MGF), and a 48-byte salt length (sLen).

```
token_authenticator_input =
  concat(Token.token_type,
         Token.nonce,
         Token.challenge_digest,
         Token.token_key_id)
valid = RSASSA-PSS-VERIFY(pkI, token_authenticator_input, Token.authenti
```

6.5. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. Each key pair SHALL be generated as as specified in FIPS 186-4 [DSS].

The key identifier for a keypair (skI, pkI), denoted token_key_id, is computed as SHA256(encoded_key), where encoded_key is a DER-encoded SubjectPublicKeyInfo (SPKI) object carrying pkI. The SPKI object MUST use the RSASSA-PSS OID [RFC5756], which specifies the hash algorithm and salt size. The salt size MUST match the output size of the hash function associated with the public key and token type.

Since Clients truncate token_key_id in each TokenRequest, Issuers should ensure that the truncated form of new key IDs do not collide with other truncated key IDs in rotation.

7. Security considerations

This document outlines how to instantiate the Issuance protocol based on the VOPRF defined in [OPRF] and blind RSA protocol defined in [BLINDRSA]. All security considerations described in the VOPRF and blind RSA documents also apply in the Privacy Pass use-case. Considerations related to broader privacy and security concerns in a multi-Client and multi-Issuer setting are deferred to the Architecture document [ARCHITECTURE].

Beyond these considerations, it is worth highlighting the fact that Client TokenRequest messages contain truncated token key IDs. This is done to minimize the chance that an Issuer can use distinct keys for targeting specific users. Since the key ID is truncated to a single byte, an Issuer can partition the set of Clients into at most 256 different anonymity sets. On top of this key ID space, Clients SHOULD apply some form of key consistency mechanism to help ensure they are not being given unique keys; see [CONSISTENCY] for more details.

8. IANA considerations

8.1. Well-Known 'token-issuer-directory' URI

This document updates the "Well-Known URIs" Registry [WellKnownURIs] with the following values.

URI Suffix	Change Controller	Reference	Status	Related information
token-issuer-directory	IETF	[this document]	permanent	None

Table 3: 'xxx' Well-Known URI

8.2. Token Type

This document updates the "Token Type" Registry from [AUTHSCHEME], Section 5.2 with the following values.

Value	Name	Publicly Verifiable	Public Metadata	Private Metadata	Nk	Nid	Reference	Notes
0x0001	VOPRF (P-384, SHA-384)	N	N	N	48	32	Section 5	None
0x0002	Blind RSA (2048-bit)	Y	N	N	256	32	Section 6	The RSABSSA-SHA384-PSS-Deterministic and RSABSSA-SHA384-PSSZERO-Deterministic variants are supported

Table 4: Token Types

8.3. Media Types

This specification defines the following protocol messages, along with their corresponding media types:

* TokenRequest: "application/private-token-request"

* TokenResponse: "application/private-token-response"

The definition for each media type is in the following subsections.

8.3.1. "application/private-token-request" media type

Type name: application

Subtype name: private-token-request

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 7

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.3.2. "application/private-token-response" media type

Type name: application

Subtype name: private-token-response

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 7

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author:
see Authors' Addresses section

Change controller: IESG

9. References

9.1. Normative References

[AUTHSCHEME] Pauly, T., Valdez, S., and C. A. Wood, "The Privacy Pass HTTP Authentication Scheme", Work in Progress, Internet-Draft, draft-pauly-privacypass-auth-scheme-00, 31 January 2022, <https://datatracker.ietf.org/doc/html/draft-pauly-privacypass-auth-scheme-00>.

[BLINDRSA] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-rsa-blind-signatures-07, 8 December 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-rsa-blind-signatures-07>.

[HTTP-Authentication] "The Privacy Pass HTTP Authentication Scheme", n.d., <https://datatracker.ietf.org/doc/html/draft-pauly-privacypass-auth-scheme-00>.

[OPRF] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-16, 28 November 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-16>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/rfc/rfc5756>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[WellKnownURIs]
"Well-Known URIs", n.d., <https://www.iana.org/assignments/well-known-uris/well-known-uris.xhtml>.

9.2. Informative References

[ARCHITECTURE] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", Work in Progress, Internet-Draft, draft-ietf-privacypass-architecture-09, 8 December 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-architecture-09>.

[CONSISTENCY] "*** BROKEN REFERENCE ***".

[DSS] "Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013, <https://doi.org/10.6028/nist.fips.186-4>.

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

Appendix A. Acknowledgements

The authors of this document would like to acknowledge the helpful feedback and discussions from Benjamin Schwartz, Joseph Salowey, Sofia Celi, and Tara Whalen.

Appendix B. Test Vectors

This section includes test vectors for the two basic issuance protocols specified in this document. Appendix B.1 contains test vectors for token issuance protocol 1 (0x0001), and Appendix B.2 contains test vectors for token issuance protocol 2 (0x0002).

B.1. Issuance Protocol 1 - VOPRF(P-384, SHA-384)

The test vector below lists the following values:

* skS: The encoded OPRF private key, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

* pkS: The encoded OPRF public key, serialized using SerializeElement from Section 2.1 of [OPRF] and represented as a hexadecimal string.

* challenge: A random challenge digest, represented as a hexadecimal string.

* nonce: The 32-byte client nonce generated according to Section 5.1, represented as a hexadecimal string.

* blind: The blind used when computing the OPRF blinded message, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

* token_request: The TokenRequest message constructed according to Section 5.1, represented as a hexadecimal string.

* token_request: The TokenResponse message constructed according to Section 5.2, represented as a hexadecimal string.

* token: The output Token from the protocol, represented as a hexadecimal string.

```
skS: 0177781aeced893dccdf80713d318a801e2a0498240fdcf650304bbbfd0f8d3b5c0
cf6cfee457aaa983ec02ff283b7a9
pkS: 022c63f79ac59c0ba3d204245f676a2133bd6120c90d67afa05cd6f8614294b7366
c252c6458300551b79a4911c2590a36
challenge:
a5d46383359ef34e3c4a7b8d1b3165778bffc9b70c9e6a60dd14143e4c9c9fbd
nonce: 5d4799f8338ddc50a6685f83b8ecd264b2f157015229d12b3384c0f199efe7b8
blind: 0322fec505230992256296063d989b59cc03e83184eb6187076d264137622d202
48e4e525bdc007b80d1560e0a6f49d9
token_request: 00011a02861fd50d14be873611cff0131d2c872c79d0260c6763498a2
a3f14ca926009c0f247653406e1d52b68d61b7ed2bac9ea
token_response: 038e3625b6a769668a99680e46cf9479f5dc1e86d57164ab3b4a569d
dfc486bf1485d4916a5194fdc0518d3e8444968421ba36e8144aa7902705ff0f3cf40586
3d69451a2a7ba210cc45760c2f1a6045134d877b39e8bcbbf920e5de4a3372557debf211
765cd969976860bc039f9082d6a3e03f8e891246240173d2cf3d69a4613b0f8415979029
22e74c7a1f2e4639e4
token: 00015d4799f8338ddc50a6685f83b8ecd264b2f157015229d12b3384c0f199efe
7b8742cdfb0ed756ea680868ef109a280a393e001d2fa56b1be46ecb31fa25e76731a5b1
d698ea7ab843b8e8a71ed9b2fffa70457a43a8fc687939424b29a7554b40fde130ab7a82
2715909cb73f99a45b640ca1c85180ba9ca1a40bab8b664406a34bcbc63b5e2e5c455cea
00001a968f7
```

B.2. Issuance Protocol 2 - Blind RSA, 2048

The test vector below lists the following values:

* skS: The PEM-encoded PKCS#8 RSA private key used for signing tokens, represented as a hexadecimal string.

* pkS: The DER-encoded SubjectPublicKeyInfo object carrying the public key corresponding to skS, as described in Section 6.5, represented as a hexadecimal string.

* challenge: A random challenge digest, represented as a hexadecimal string.

* nonce: The 32-byte client nonce generated according to Section 6.1, represented as a hexadecimal string.

* blind: The blind used when computing the blind RSA blinded message, represented as a hexadecimal string.

* salt: The randomly generated 48-byte salt used when encoding the blinded token request message, represented as a hexadecimal string.

* token_request: The TokenRequest message constructed according to Section 6.1, represented as a hexadecimal string.

* token_request: The TokenResponse message constructed according to Section 6.2, represented as a hexadecimal string.

* token: The output Token from the protocol, represented as a hexadecimal string.

skS: 2d2d2d2d2d424547494e205052495641544520b45592d2d2d2d2d0a4d494945765
149424144414e42676b71686b69473977304241514546414153424b63776767536a416
74541416f49424151444c4775317261705831736334420a4f6b7a38717957355379356b6
f6a41303543554b66717444774e38366a424b5a4f76457245526b49314c527876734d645
3327961326333616b4745714c756b440a556a35743561496b317241764365584464644e445
0344232505570785143 6e6969396e6b492b6d67725769744444449487138613979313758 6
e6c5079596f784f530a646f6558563835464f314a752b62397336356d586d34516a75513
945596149713833717244505 67a50335758712b524e4d636379323269686763624c766d4
2390a6a41355334475666325a6c74785954736f4c364872377a58696a4e3946374862716
5676f753967654b524d584645352f2b4a3956595a634a734a624c756570480a544f72535
a4d4948502b5358514d4166414f454a4547426d6d4430683566672f43473475676a79486
e4e51383733414e4b6a55716d367657457441 3872514c620a4530742b496c706641674d4
2414145436767454 14c7a4362647a69316a506435384d6b562b434c6679665351322b726
6486e7266724665502f566344787275690a3270316153584a596962653645532b4d622f4
d4655646c485067414c77317851345765726636633644437368 6c6c784c5753563847734
2737663386f364750320a6359366f777042447763626168474b556b5030456b623953305
84c4a5763475347356155 6e484a585237696e7834635a6c666f4c6e7245516536685578 7
34d710a623064487864484442 4d64476656577767 4b6f6a4f6a70532f39386d455579375
6422f3661326c7265676c766a632f326e4b434b7459373744376454716c47460a787a414
261577538364d435a342f5131334c762b42656662717449397 3715a5a776a72645568514
838564378727932 51564d515751696e57684174364d7154340a53425354726f6c5a7a777
2716a65384d504a393175614e4d6458474c63484c49323673587a76374b53514b4267514
47663777350555576 41395a325a583958350a6d49784d54424e6445467a5662555 0754b4
b413179576e31554d444e63556a71682b7a652f376b337946786b68305146333162713 63
0654c393047495369414f0a354b4f574d39454b6f2b784151326 2614b314d664f5931472
b386a7a42585570427339346b353353383879586d4b366e7964677637304 24a385a68356
66b55710a5732306f5362686b686a5264537a48326b52476972672b5553774b426751445
a4a4d6e7279324578612f3345713750626f737841504d69596e6b354a415053470a79327
a305a375455622b7548514f2f2b78504d376e433075794c494d44396c61544d48776e367
3372f4c62476f4550315752677 06f59482f4231346b2f526e360a667577524e3632496f3
97463392b41434c745542377674 476179332b675277597453433262356564386c4969656
774546b656130683075445352784174567 3330a6e356b796132513976514b4267464a754
67a4f5a742b7467596e576e5155456757 3850304f494a45484d45345554644f637743784
b7248527239334a6a7546320a453377644b6f546969375 072774f59496f614a5468706a5
0634a62626462664b792b6e7351703159477639776 44a724d6156774a6376497077 56367
6315570660a56744c61646d316c6b6c767 0717336474e4d386a6e4d30587833616a6d6d6
e66655739794758453570684d727a4c4a6c394630396349324c416f4742414e58760a756
7565872703262735 4316f6b6436755361427367704a6a5065774e526433635a4b397a306
153503144544131504e6b7065517748672f2b36665361564f487a0a79417844473396835 5
2726278526 14e6673542b7241554837783153594456565 159564d68555262546f5a65364
72f6a716e544333664e66485631787 45a666f740a306c6f4d4867776570362b53494d436
f6565325a6374755a5633326c634961663972 62484f633764416f47416551386b3853494
c4e4736444f413331544535500a6d3031414a495977373416c5233756f2f524e61432b785
964505533547 36b75414c78786944452 2f57734c455142436a6b46576d6d4a41576e51554
474626e594e0a536377523847324a36466e7245437462747973373357415 6476f6f465a6
e636d504c50386c784c79626c534244454c79615a762f624173506c4d4f39624435630a4
a2b4e534261612b6f694c6c31776d4361354d43666c633d0a2d2d2d2d2d454e442050524
956415445204b45592d2d2d2d2d0a
pkS: 30820152303d06092a864886f70d01010a3030a00d300b0609608648016503040 20
2a11a301806092a864886f70d010108300b0609608648016503040202a20302013003820
10f003082010a0282010100cb1aed6b6a95f5b1ce013a4cfcab25b94b2e64a23034e4250
a7eab43c0df3a8c12993af12b111908d4b471bec31d4b6c9ad9cdda90612a2ee903523e6
de5a224d6b02f09e5c374d0cfe01d8f529c500a78a2f67908fa682b5a2b430c81eaf1af7
2d7b5e794fc98a3139276879757ce453b526ef9bf6ceb99979b8423b90f4461a22af37aa
b0cf5733f7597abe44d31c732db68a181c6cbbe607d8c0e52e0655fd9996dc584eca0be8
7afbcd78a337d17b1dba9e828bbd81e291317144e7ff89f55619709b096cbb9ea474cead
264c2073fe49740c01f00e109106066983d21e5f83f086e2e823c879cd43cef700d2a352
a9babd612d03cad02db134b7e225a5f0203010001

```
challenge:
3f5a1c30d13f860622458ce836d8af325378054370fe8a3d771eebd67d4d810d
nonce: c0fcbbb243d8f5d4f661dbdefca95879b39aeccb77b7db731b59c09688773125
blind: 04d00c700128b4b201b4bec4f05d942bc903d49c26568b5956e0827590d2e4b43
570105ae492f655d41a3d68f1cc6a9a2895c36fd45c88239257f2e6cee5bd88e7d870f35
67069d78f8b85947c7ab123b16c9f3b76d856112802dd0fefa800a9c3807fbb5d949481b
4f7a21da0269f17611b93dfa7197e87ef9c1ef9c2fd0f86119917cdf01284038435f2df3
f8ae2935ae0ef5440b3b4ac12fed83a03bc494abaa87241d624d2dcb0c6a64422eb63dbf
ba0193161648e5b2afbdf3140901840c7d08a0e2953320fffa09641500122ba81c5907e7
ebd4d2384221ddb99439c2465138b98348b58a5f89b4e05b70856a270e1f5308512e368c
fe6dfe4cf3759ed
salt: 4daf07bc96a829736ce6386a4d3ed988192ea4f0acb3ed715dca2ae688c16ad346
ee5e2b3dd26eb2868639a778e3bc5d
token_request: 0002ca832fffabdd44e2cd54e5e24d74519d297608aec9ab88e26b732
adcb382781e7e2657c8b94751b9fa6b2ed02cec383f8cd04e9627d5b62a7f1b7ea16b81e
46f35637cca49f8990d5359f8a7dcac1ba58fb685d4b32a67621d368cc112197d4f84ee5
241c359299cb5fc41182bd65bba112f35a4073d1231290447fb884888ba84eb5b4602534
787aa1e167bc1ddcceb7fb5ad43e2b242fd4b4939349897cfb911cf0f3785847edaeaca6
350c16cb05b7882ec076a3adde7c361f54d6eb67ec239aeafe8a4816b29e4c6aa8bf2873
ba36ec6e2b9596aa508b5e34543a469286be2404f1f481f6a274a2afb429d62377f7ab6d
e56379d2c42f7205e3bf1c74d3159
token_response: 6e7d5334765bea44ea43b81ae8f41334fdac47b3dfaaeb2c3b99f42a
67d8239592ac4fa129a938e139bf052d85804bdaa90f7f54fdfa34d6efeaea0ccc15a500
fb2987b534d0558e8d32df68b3533f6cbc953dabcfff2ef6b6af336c1128f607f0796190
6a2fed919691340e751a8e2173e674569f7e4beb7ad0ee5c65ce82ad477d3e44b3755bcd
0f168ab85ce662d3f87c5634be036382d6ad4ab870ab975e8bffd0b95bcf457dc83337ff
ea85b7c77d44e5cb4bddc5aecfc958cc822cc53ded3da699af86bfad3054fe49da8eeb55
162e444a3b4d438f9e3cbadd50cba56b4f3f0718a65e7d8dfc40762cdb9962edc731f6a7
ec8641cbf98a0ec9cdf8b7f6
token: 0002c0fcbbb243d8f5d4f661dbdefca95879b39aeccb77b7db731b59c09688773
125ad76ab53adc2ca44e4eaae3d71b9bf3fc9332122faeef07cb70d9e04da68c6a7ca572
f8982a9ca248a3056186322d93ca147266121ddeb5632c07f1f71cd27080d1f816364e5d
4d516d2f3e80366e56edc1de4ba0d7aed2675c15156d774b311778091bf5f2aea9926156
2289459a41c5739dec6dc42447744fe07c53c9d090f053263d019255cdfc27739132bd68
21ad49f1a98db6873319d04c04703d74a8fe1d0806b2a25b46246c5bb2ff927463b03152
589068389df89494c6d82f3b92be773a9fe6bc1fed9cbf26bdfbae1ff369f20d0267cdd2
0f3bcba30f8b0c0e9d9a1a39a40156b0614030d5099aa36f085347681aef502f3d081b36
cd79f7ea14df1ca9694320fc44ccbc7c5d90aeedc915af3ac11a3baf562d38c8213e39f6
731fa5e701697d0bfbfcfc83b447945b351115a20770370226b52a19df939f3080e
```

Authors' Addresses

Sofía Celi
Brave Software
Lisbon
Portugal

Email: cherenkov@riseup.net

Alex Davidson
Brave Software
Lisbon
Portugal

Email: alex.davidson92@gmail.com

Armando Faz-Hernandez
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: armfazh@cloudflare.com

Steven Valdez
Google LLC

Email: svaldez@chromium.org

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net